big O (informal)

In the next several lectures, we will study how the number of operations performed by various algorithms grows with n which is the size of the input. We have seen several examples of this growth.

- addition grows linearly with the number of digits in the two numbers (assuming they have the same number of digits). We say that the addition algorithm is $O(n)$.
- Multiplication grows with the square of the number of digits. We say this algorithm is $O(n^2)$. The reason it is n^2 is that the algorithm involves the single digit products of each possible *pair* of digits in the first and second numbers and there are n^2 pairs.
- Insertion sort (worst case) grew as n^2 so we say it is $O(n^2)$. The reason it is n^2 is that the algorithm involves two loops, one nested inside the other, which leads to 1 + 2 + ... + n operations.
- Insertion sort (best case) grew as n so we say it is $O(n)$. The best case occurs if the list is already sorted. In this case the inner loop only executes a small number of operations (not dependent on n).
- (coming up) Mergesort is $O(n \log n)$. The merge steps at each "level" involve a total of n operations and there are $\log n$ levels, since the number of levels is the number of times we divide n by 2 before we reach 0. (It is more challenging to visualize this.)
- (coming up) Binary search is $O(\log n)$. It involves $\log n$ recursive calls (for the same reason mergesort has $\log n$ levels).

We have seen several algorithms in the course, and loosely characterized the time it takes to run them in terms of the "size" n of the input. Let's now tighten up our analysis. We will study the behavior of algorithms by comparing the number of operations required – which is typically a complicated function of n of the input size n – against some simpler function of n. This simpler function describes either an upper bound, in a certain technical sense to be specified below, or a lower bound, or both.

Limits In Calculus, you look at the limit of a sequence a_n as $n \to \infty$. For example, if $a_n = 7+2^{-n}$, then the limit is 7. Formally, the limit is defined as follows: We say $\lim_{n\to\infty} a_n = a$ if for any $\epsilon > 0$, there exists and n_0 such that if $n > n_0$ then $|a_n - a| < \epsilon$.

Big O

Let $t(n)$ be a well-defined sequence of numbers. This sequence $t(n)$ represents the "time" or number of steps it takes an algorithm to run as a function of some variable n which itself represents the "size" of the input. We will consider $n \geq 0$ and $t(n)$ to both be positive and real.

Let $g(n)$ be another well defined sequence of reals. We commonly consider $g(n)$ to be one of the following:

$$1, \log n,\ n,\ n\log n,\ n^2,\ n^3,\ 2^n, \ldots$$

We would like to say that $t(n)$ is bounded above by a simple $g(n)$ function if, for n sufficiently large, we have $t(n) \leq g(n)$. We would say that $t(n)$ is "asymptotically bounded above" by $g(n)$. Formally, let's say that $t(n)$ *is asymptotically bounded above by* $g(n)$ if there exists a positive number n_0 such that, for all $n \geq n_0$,

$$t(n) \ \leq g(n).$$

For example, consider the function $t(n) = 5 + 7n$. For n sufficiently large, $t(n) \leq 8n$. Thus, $t(n)$ is "asymptotically bounded above" by $g(n) = 8n$ in the sense that I just define. Notice that the constant 8 is arbitrary here. Any constant greater than 7 would do. For example, $t(n)$ is also "asymptotically bounded above" by $g(n) = 7.00001n$.

It is much more common and useful to talk about asymptotic upper bounds on $t(n)$ in terms of a simpler function $g(n)$, namely where we don't have constants in $g(n)$. This is justified by the fact that exact constants are generally implementation dependent. We typically care about the growth of the function but not precise constants. To do this, one needs a slightly more appropriate definition. This is the standard definition of an asymptotic upper bound:

Definition (big O): The sequence $t(n)$ is $O(g(n))$ if there exists two positive numbers n_0 and c such that, for all $n \geq n_0$,

$$t(n) \ \leq \ c \, g(n).$$

We say $t(n)$ is "big-O of $g(n)$".

I emphasize that the condition $n \geq n_0$ allows us to ignore how $t(n)$ compares with $g(n)$ when n is small. In this sense, it describes an *asymptotic* upper bound.

Example 1

The function $t(n) = 5 + 7n$ is $O(n)$. To prove this, we write:

$$\begin{aligned} t(n) &= 5 + 7n \\ &\leq 5n + 7n, \quad n \geq 1 \\ &= 12n \end{aligned}$$

and so $n_0 = 1$ and $c = 12$ satisfy the definition. An alternative proof is:

$$\begin{aligned} t(n) &= 5 + 7n \\ &\leq n + 7n, \text{ for } n \geq 5 \\ &= 8n \end{aligned}$$

and so $n_0 = 5$ and $c = 8$ also satisfy the definition.

A few points to note:

- If you can show $t(n)$ is $O(g(n))$ using constants c, n_0, then you can always increase c and/or n_0 and be sure that these constants will satisfy the definition also. So, don't think of the c and n_0 as being uniquely defined.
- There are inequalities in the definition, e.g. $n \geq n_0$ and $t(n) \leq cg(n)$. Does it matter if the inequalities are strict or not? Not really. You can easily verify that, for any $t(n)$ and $g(n)$, the statement "$t(n)$ is $O(g(n))$" is true (or false) whether we have a strict inequality or just "less than or equal to".

- We are looking for tight upper bounds for our $g(n)$. But the definition of big O doesn't require this. For example, in the above example, $t(n)$ is also $O(n^2)$ since $t(n) \leq 12n \leq 12n^2$ for $n \geq 1$. By a similar argument, $t(n)$ is also $O(n \log n)$ or $O(n^3)$, etc.

Many students write "proofs" that they believe are correct, but in fact the "proofs" are incomplete (or wrong). For example, consider the following "proof" for the above example:

```
 5 + 7n   <  c n
5n + 7n   <  c n, n >= 1
    12n   <  c n

Thus,  c > 12,  n_0 = 1
```

There are several problems with this, as a formal proof.

- the first statement seems to assume what we are trying to prove; some indication should be given as whether this first line is "an assumption", or this is "what we are going to show". And is the statement true for *some* `c`, or for *all* `c`, or for *some* `n`, or *all* `n`? Its just not clear.

- There is no clear connection between the statements. What implies what? Are the statements equivalent, etc ? *Such proofs may get grades of 0. This is not the "big O" you want.*

What would it mean to say $t(n)$ is $O(1)$, i.e. $g(n) = 1$? Applying the definition, it would mean that there exists a c and n_0 such that $t(n) \leq c$ for all $n \geq n_0$. That is, $t(n)$ is bounded by some constant.[11]

Example 2 (see a different example in slides)

The function $t(n) = 17 - 46n + 8n^2$ is $O(n^2)$. To prove this, we need to show there exists positive c and n_0 such that, for all $n \geq n_0$,

$$17 - 46n + 8n^2 \leq cn^2 \ .$$

$$\begin{aligned} t(n) &= 17 - 46n + 8n^2 \\ &\leq 17 + 8n^2, \quad n > 0 \\ &\leq 17n^2 + 8n^2, \text{ if } n \geq 1 \\ &= 25n^2 \end{aligned}$$

and so $n_0 = 1$ and $c = 25$ do the job.

We can perform an alternative manipulation:

$$\begin{aligned} t(n) &= 17 - 46n + 8n^2 \\ &\leq 17 + 8n^2, \quad n > 0 \\ &\leq n^2 + 8n^2, \textit{ if } n \geq 5 \\ &= 9n^2 \end{aligned}$$

and so $c = 9$ and $n_0 = 5$ do the job as well.

[11]Why would this be used? Sometimes when we analyze an algorithm's performance, we consider different parts of the algorithm separately. Some of those parts (such as assigning values to variables outside of any loops or any recursive calls) might be done once. Such parts take $O(1)$ time.

Example 3

Show $t(n) = \frac{500+20\log n}{n}$ is $O(1)$.

First note

$$t(n) = \frac{500 + 20\log n}{n} \leq \frac{500 + 20n}{n}$$

since $\log n < n$ for all $n \geq 1$ (see below). We now want to show there exist positive c and n_0 such that for all $n \geq n_0$,

$$\frac{500 + 20n}{n} \leq c$$

Take $c = 520$. Then we want to show there exists an n_0 such that

$$500 + 20n \leq 520n$$

for all $n \geq n_0$. But $n_0 = 1$ clearly does the job.

[BEGIN ASIDE: Can you formally prove the "obvious" claim that $\log n < n$ for $n \geq 1$. One easy way to do this is to note the "even more obvious" claim $n < 2^n$ for all $n \geq 1$ and take the log of both sides (because log is monotone and increasing). To formally *prove* that $n < 2^n$ for $n \geq 1$, use induction. The base case is $1 < 2^1$. For the induction step, we have for $k \geq 1$,

$$\begin{aligned} k+1 &< 2^k + 1 \text{ by the induction hypothesis} \\ &< 2^k + 2^k \\ &= 2^{k+1} \text{ and we are done.} \end{aligned}$$

END ASIDE]

Subset notation

For any $g(n)$, we *can define* $O(g(n))$ *as the set of functions* $t(n)$ that satisfy the above definition. In this interpretation, we would say $t(n) \in O(g(n))$ rather than $t(n)$ "is" $O(g(n))$. With this set interpretation, one can formally show (though this is not obvious yet) that

$$O(1) \subset O(\log n) \subset O(n) \subset O(n\log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \cdots \subset O(n!)$$

for example, if a function $t(n)$ is $O(\log n)$ then it automatically is $O(n)$, etc. (The reason comes down to the fact that $\log n$ is itself $O(n)$.)

Some background logic

The first thing to appreciate is that when you have multiple "for all" and "there exists" quantifiers in a single statement, the order matters. For example, suppose I say that "for all integers x, there exists an integer y such that $y > x$". This statement is true; it just means that there is no largest integer. However if I reverse the quantifiers, then I get the statement "there exists an integer y such that for all integers x, $y > x$"; it just means that there exists an integer larger than any integer, which is now false.

Now, let's look at the problem of how to negate statements that involve logical quantifiers.

Consider the statement "$t(n)$ *is asymptotically bounded above by* $g(n)$ if there exists a positive number n_0 such that, for all $n \geq n_0$, $t(n) \leq g(n)$." How do we negate such a statement? We say $t(n)$ is *not* asymptotically bounded above by $g(n)$ if there *does not* exist a positive number n_0 such that, for all $n \geq n_0$,

$$t(n) \leq g(n).$$

This means that for all $n_0 \geq 0$ we may choose, there exists an $n \geq n_0$ such that $t(n) > g(n)$. Each "there exists" becomes a "for all" and vice versa, and the final statement is negated.

not big O

What does it mean for a function $t(n)$ *not* to be $O(g(n))$, or equivalently, that that statement "$t(n)$ is $O(g(n))$" is false? Saying $t(n)$ is *not* $O(g(n))$ means that there do *not* exist two positive constants c and n_0 such that, for all $n \geq n_0$, $t(n) \leq cg(n)$. Logically, this is equivalent to saying that, *for all* two positive constants c and n_0, there exists an $n > n_0$ such that $t(n) > cg(n)$. Let's look a few examples of $t(n)$ and $g(n)$ for which $t(n)$ is not $O(g(n))$.

Example: $t(n) = 3^n$ **is** *not* $O(2^n)$

Take any $c > 0$ and $n_0 \geq 0$. We want to show there exists an $n > n_0$ such that $3^n > c2^n$, or equivalently, $(\frac{3}{2})^n > c$. Clearly such an n exists since the left hand side of the last inequality increases without bound, whereas the right hand side is a constant. To find a value of n, take logs of both sides and manipulate to get $n > \log_{\frac{3}{2}} c$. Thus, if we choose any n greater than $\max(n_0, \log_{\frac{3}{2}} c)$ then $3^n > c2^n$. We have therefore shown that $t(n) = 3^n$ is *not* $O(2^n)$.

Example: $t(n) = 3n^2 + 5n + 2$ **is** *not* $O(n)$

To prove formally that $t(n)$ is not $O(n)$, we use a different argument from what we saw last lecture, namely we use a *proof by contradiction.* We hypothesize that $t(n)$ is indeed $O(n)$, and then we show that this hypothesis cannot be correct, i.e. it leads to a contradiction.

Assume $t(n)$ is $O(n)$. Let $c > 0$ and $n_0 \geq 0$ be two constants such that, for all $n \geq n_0$,

$$3n^2 + 5n + 2 \leq cn.$$

Dividing both sides by n gives an equivalent inequality,

$$3n + 5 + \frac{2}{n} \leq c \ . \qquad (*)$$

But we see that this cannot be true for all $n \geq n_0$ since the left side grows without bound.

To get a concrete value of the n that produces the failure, note that

$$\begin{aligned} 3n + 5 + \frac{2}{n} &> 3n + 5, \quad \text{if } n > 0 \\ &\geq c, \quad \text{if } n \geq max\{0, \frac{c-5}{3}\} \end{aligned}$$

So, any $n > max(\frac{c-5}{3}, n_0)$ will contradict the big O definition for the chosen constants n_0, c. Hence it is *not* possible to find such constants that satisfy the big O definition. Hence $t(n)$ is not $O(n)$.

In the proof above, do you need to provide the "concrete value" of n? Or could you stop at the line (*) ? In this case, I would say the latter is fine since there is nothing very subtle about the claim that the left side increases without bound as $n \to \infty$.

Big Omega (asymptotic lower bound)

With big O, we defined an asymptotic upper bound. There is a similar definition for asymptotic lower bounds. This lower bound is used to claim that something can only be done so fast. You will make great use of $\Omega()$ arguments in future Algorithms courses.

As an example, consider $t(n) = \frac{n(n-1)}{2}$. For sufficiently large n, this function is greater than $g(n) = \frac{n}{4}$. More formally, we would say that $t(n)$ is asymptotically bounded below by $g(n)$ if there exists an n_0 such that, for all $n \geq n_0$, we have $t(n) \geq g(n)$. The definition of $\Omega(\ )$ is slightly more complicated than this, though, since we want to use functions $g(n)$ that don't have constants in them – just like in the definition of $O(\ )$.

Definition (big Ω): We say that $t(n)$ is $\Omega(g(n))$ – "big Omega of $g(n)$" – if there exists positive constants n_0 and c such that, for all $n \geq n_0$,

$$t(n) \geq c\, g(n).$$

The idea is that $t(n)$ grows at least as fast as $g(n)$ times some constant, for sufficiently large n. Note that the only difference between the definition of $O()$ and $\Omega()$ is the $\leq$ vs. $\geq$ inequality.

As an example, you will prove in COMP 251 that the average case behavior, $t(n)$, of *any* sorting algorithm is $\Omega(n \log n)$. That is, it is impossible that algorithm that sorts n arbitrary numbers using fewer than $cn \log n$ operations, on average.[12]

See the slides for a few examples and see Exercises 5 for many more examples.

Big Theta

It often happens that $t(n)$ is both $O(g(n))$ and $\Omega(g(n))$. For example, $t(n) = \frac{n(n+1)}{2}$ is both $O(n^2)$ and $\Omega(n^2)$. In this case, we say that $t(n)$ is $\Theta(g(n))$, i.e. "big theta".

[12]If you are wondering how "average" is defined here, note that there are $n!$ arrangments of n distinct numbers and only one of the arrangements is correctly sorted. "Average case" analysis treats the $n!$ arrangements as equally likely.

Subset notation

Similarly to big-O set notation, we can write

$$\cdots \subset \Omega(n^2) \subset \Omega(n \log n) \subset \Omega(n) \subset \Omega(\sqrt{n}) \subset \Omega(\log n) \subset \Omega(1)$$

That is, $\Omega(1)$ is the largest of the sets shown. It is the set of functions that are asymptotically bounded below by a positive constant – note that this includes any function that converges above 0 as $n \to \infty$.